Today’s Topics

Lexing & Parsing
Grammars
Regular Expressions
State Machines: Formalism

Formally, a state machine consists of

- A set of states, State = {…}
- A set of initial states, Init = {…}
- A set of events, Event = {…}
- A transition relation trans ⊆ State × Event × State
- A set of traces (derived from trans and Init)
MylIterator State Machine

**Mutable object can be modeled by state machine**

- State is represented by instance variables of the object
- Change in state = changing instance variables
- Events are operations that can be performed on the object—its methods
- Mutators change state

![State Machine Diagram]

- INPROGRESS
  - hasNext/true
  - next/element
- DONE
  - hasNext/false
  - next/element
Running Examples: Markup

We’ll consider 3 different (simplified!) markup languages

**HTML**

- This is an `<i>`italic`</i>` word.

**Markdown**

- This is _italic_.

**LaTeX**

- LaTeX considers `{em italics as emphasis, though `{em nested emphasis}` results in un-italicizing}. 
State Machine for Markup

Let’s build a state machine for our markup languages.
State Machine for Markup

Let’s build a state machine for our markup languages.
State Machine for Markup

Let’s build a state machine for our markup languages.

State machine can’t express structure of LaTeX’s nesting, or if we allow nesting in HTML.
Building Markup Interpreters

Let’s build some code to read & interpret our markup languages.

Lexical Analysis
- Transforms a stream of characters into higher-level symbols
- Also called “lexing”; results called “tokens” or “lexemes”

Parsing
- Interprets the stream of higher-level symbols
- Responsible for understanding the relationships between tokens
- E.g. for Markdown, responsible for matching _’s

Usually lexing & parsing done by separate state machines.
Building Markup Interpreters

**Though lexer & parser are separate, they communicate**

- Parser consumes tokens from lexer
- Closely coupled, but clear contract between them

**Example of Separation of Concerns**

- Software engineering principle that allows independent development
- Communication through a clear interface/contract
- Prevents any single piece from doing too much
Lexing/Tokenization

Goal: aggregate input stream of characters into higher-level symbols.

```c
int square(int x) {
    return x * x;
}
```

Example token stream

- `int, square, (, int, x, ), {, return, x, *, x, ;, }`

Usually, grouped into kinds of token

- `int, id("square"), (, int, id("x"), ), {, return, id("x"), *, id("x"), :, }`
- Returned as objects with fields
Tokenization is Language-dependent

Lexing Java is different from lexing Python

- Java ignores whitespace, while Python requires preserving indentation
- So Python lexing will contain information about indentation & newlines

Lexer Interface

- Similar to an iterator: hasNext() and next() returns a token
- Alternatively, special token to signal end: END or throw exception
Lexing our Markup

**Tokens for our HTML subset**

- 3 token types: text, `<i>`, and `</i>`
- E.g. This is an `<i>italic</i>` word.
  - text(“This is an”), `<i>`, text(“italic”), `</i>`, text(“ word.”)

**Tokens for Markdown**

- 2 token types: text, `_`
- E.g. This is an `_italic_` word.
  - text(“This is an”), `_`, text(“italic”), `_`, text(“ word.”)

**Tokens for LaTeX**

- 3 token types: text, `{\em, }`
- E.g. LaTeX supports `{\em italics with {\em nesting}}`.
  - text(“LaTeX supports”), `{\em`, text(“ italics with “), `{\em`, text(“nesting”), `}`, `}`
Parsing & Grammars

Define a set of sentences that are considered valid sequence of symbols/tokens (aka “terminals” in grammar parlance).

Grammars define the language to parse.

Structure

- Grammar is a set of productions
- Each production defines a non-terminal
- Non-terminal is a variable that stands for a set of sub-sentences
Grammars

Backus-Naur Form (BNF)

- non-terminal ::= expression of terminals, non-terminals, and operators
- Non-terminals are capitalized, terminals are lower-case

Basic Operators

- Sequence: A ::= B C
  - i.e. A is a B followed by C
- Iteration: A ::= B*
  - i.e. A is zero or more B’s
- Choice: A ::= B | C
  - i.e. A is either B or C
Grammars: More Operators

Additional operators provide useful syntactic sugar.

Common additional operators

- **Grouping:** $A ::= (B \ C)^*$
  - $A$ is zero or more $B$ $C$ pairs

- **1+ iteration:** $A ::= B^+$
  - Same as $A ::= BB^*$

- **Character classes**
  - $A ::= [abc]$ is the same as $A ::= a | b | c$
  - $A ::=[^b]$ is the same as $A ::= a | c | d | e | …$
Grammar Example: Markdown

Given the operators we’ve covered, let’s now define a grammar for our subset of Markdown.

Markdown ::= ( Normal | Italic ) *
Italic ::= _ Text _
Normal ::= Text
Text ::= [^ _ ]*

Building blocks
➢ Terminals: _, strings of text without _
➢ Non-terminals: Markdown, Italic, Normal, Text

This grammar does NOT allow nested italics.
Grammar Example: HTML Subset

Html ::= ( Normal | Italic ) *
Italic ::= <i> Html </i>
Normal ::= Text
Text ::= [^ < ]*

Are nested italics allowed in this grammar?
Grammar Example: LaTeX Subset

\[
\text{LaTeX ::= ( Normal | Italic ) } *
\]
\[
\text{Italic ::= } \{ \text{em } \text{LaTeX } \} 
\]
\[
\text{Normal ::= Text} 
\]
\[
\text{Text ::= [^ {} ]}* 
\]

This grammar allows nesting of italic regions.
More Complicated Example: URLs

URL ::= Protocol :// Address
Address ::= Domain . TLD
Protocol ::= http | ftp
Domain ::= mit | apple | pbs
TLD ::= com | edu | org

**Terminals**
- ://, ., http, ftp, mit, apple, pbs, com, edu, org

**Non-terminals**
- TLD = {com, edu, org}
- Domain = {mit, apple, pbs}
- Protocol = {http, ftp}
- Address = {mit.com, mit.edu, mit.org, apple.com, apple.edu, apple.org, pbs.com, pbs.edu, pbs.org}
Regular Grammars

Markdown grammar seems different: it is a regular grammar.

Property of Regular Grammars

By substituting every non-terminal except root with its right hand side, can be reduced to a single production from the root, with only terminals and operators on the right hand side.

Markdown ::= ( Normal | Italic ) *
Italic ::= _ Text _
Normal ::= Text
Text ::= [^ _ ]*
Regular Grammars

Markdown grammar seems different: it is a regular grammar.

Property of Regular Grammars
➢ By substituting every non-terminal except root with its right hand side, can be reduced to a single production from the root, with only terminals and operators on the right hand side.

Markdown ::= ( Normal | _ Text _ ) *
Normal ::= Text
Text ::=[^ _ ]*
Regular Grammars

Markdown grammar seems different: it is a regular grammar.

Property of Regular Grammars

By substituting every non-terminal except root with its right hand side, can be reduced to a single production from the root, with only terminals and operators on the right hand side.

Markdown ::= ( Text | _ Text _ ) *
Text ::= [^ _ ]*
Regular Grammars

Markdown grammar seems different: it is a regular grammar.

Property of Regular Grammars

- By substituting every non-terminal except root with its right hand side, can be reduced to a single production from the root, with only terminals and operators on the right hand side.

Markdown ::= ([^_]* | _[^_]* _ ) *

This is called a regular expression.
Regular Expressions

So useful, most modern programming languages implement them in standard libraries.

Java supports for regexes

- `String.split()`, `String.match()`, `java.util.regex`, & more
- Useful & commonly-used for manipulating strings

Major operations

- `match`: does this string match a regex
- `split`: return an array of strings by splitting the original around the regex
- `replace`: replace a substring matching a regex with a new string
Regular Expressions

Most languages use Perl-derived syntax for regular expressions

- Includes more operations than those we’ve discussed
- Major addition: Character “classes” such as whitespace, uppercase, etc
  - \s for whitespace, \d for digit, etc

Example (what does this do?)
String m = ...
m.replace("\s+", "\t\n").replace("^\s+", "\t\n").replace("\s+$", "\t\n");

Big caveat: Regexes can be confusing to understand!
Regular Expressions

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM WE’D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

IT’S HOPELESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

http://xkcd.com/208/
Hierarchy of Grammars

Most programming languages can be expressed using our grammars. Such languages are called context-free grammars.

Not all context-free languages are regular
- HTML & LaTeX are context-free but not regular
- Java itself is context-free & not regular

Intentionally being hand-wavy
- For details, take 6.045!
Regular Grammars & State Machines

Regular grammars can be parsed by a special kind of state machine: a non-deterministic finite state machine (NFSM or NFA for “non-deterministic finite automaton”)

Usually powerful enough for lexing.

Non-regular grammars require more computational power

➢ Intuition: if set of states is finite, and no other way to record anything, not enough “memory” for things like nesting

More info in 6.045 or Sipser’s *Introduction to the Theory of Computation*
Parsing Non-Regular Grammars

A simple recipe for turning recursive grammars into code

- Called recursive-descent parsing
- Does not work for all grammars, but does for an important sub-class of context-free grammars
- Alternative: use a parser-generator like YACC or JavaCC or ANTLR

Recursive-Descent Parsing

1. Treat the parser as a state machine that incrementally consumes tokens from the lexer
2. For a sequence of tokens, look at the beginning of the token stream to determine which case of the production applies
3. Given the case, parse each of the terminals & non-terminals in order
   - Terminals parsed by consuming the token
   - Parsing a non-terminal: make a recursive call to the associated function
A Recursive-Descent Parser for Markdown

Markdown ::= ( Normal | Italic ) *
Italic ::= _ Text _
Normal ::= Text
Text ::= [^ _ ]*

Methods
➢ evalMarkdown(), evalNormal(), evalItalic()
➢ Why not evalText()?
A Recursive-Descent Parser for Markdown
Summary

Machine processing of text is important in many areas of computer science.

Context-free languages describe most programming languages & regular languages are an important subset that can be described without recursion.

Processing languages done using combination of lexer & parser to separate concerns.